Effect-driven QuickChecking of compilers

Midtgaard, Jan; Justesen, Mathias Nygaard; Kasting, Patrick Frederik Soelmark; Nielson, Flemming; Nielson, Hanne Riis

Published in:
Proceedings of the Acm on Programming Languages

Link to article, DOI:
10.1145/3110259

Publication date:
2017

Document Version
Publisher's PDF, also known as Version of record

Effect-Driven QuickChecking of Compilers

JAN MIDTGAARD, Technical University of Denmark, Denmark
MATHIAS NYGAARD JUSTESEN, Technical University of Denmark, Denmark
PATRICK KASTING, Technical University of Denmark, Denmark
FLEMMING NIELSON, Technical University of Denmark, Denmark
HANNE RIIS NIELSON, Technical University of Denmark, Denmark

How does one test a language implementation with QuickCheck (aka. property-based testing)? One approach is to generate programs following the grammar of the language. But in a statically-typed language such as OCaml too many of these candidate programs will be rejected as ill-typed by the type checker. As a refinement Pałka et al. propose to generate programs in a goal-directed, bottom-up reading up of the typing relation. We have written such a generator. However many of the generated programs has output that depend on the evaluation order, which is commonly under-specified in languages such as OCaml, Scheme, C, C++, etc. In this paper we develop a type and effect system for conservatively detecting evaluation-order dependence and propose its goal-directed reading as a generator of programs that are independent of evaluation order. We illustrate the approach by generating programs to test OCaml's two compiler backends against each other and report on a number of bugs we have found doing so.

CCS Concepts: • **Software and its engineering → Software testing and debugging**; • **Theory of computation** → *Program analysis*; *Operational semantics*;

Additional Key Words and Phrases: QuickCheck, compiler testing, type and effect system

ACM Reference Format:
Jan Midtgaard, Mathias Nygaard Justesen, Patrick Kasting, Flemming Nielson, and Hanne Riis Nielson. 2017. Effect-Driven QuickChecking of Compilers. *Proc. ACM Program. Lang.* 1, ICFP, Article 15 (September 2017), 23 pages.
https://doi.org/10.1145/3110259

1 INTRODUCTION

Consider the following effectful OCaml program:

```
let k =
  (let i = print_newline ()
   in fun q -> fun i -> "") ()
in 0
```

The program contains a subtle combination of a partially applied function and a side effect. When run the program should output a newline character and return 0. Interestingly OCaml's native code backend does not compile it correctly: it produces code which erroneously delays the side effect (in this case indefinitely). We found this bug (and others) by generating arbitrary effectful OCaml

2475-1421/2017/9-ART15
https://doi.org/10.1145/3110259

programs, compiling each of them with OCaml's two backends, and testing that the behaviour of the produced code agree.

How does one generate such arbitrary programs to test the backend of a language implementation? Simply generating arbitrary text strings will not lead to much backend testing as almost all of the strings will be rejected by the parser. Nor will an approach based on generating arbitrary abstract syntax trees as too many of those will be rejected by the type checker. Instead one should take the approach of Pałka et al. [2011] and generate type correct programs following a goal-directed, bottom-up reading of the type system. However, suppose the language implementation is loosely defined, meaning that the implementation is not fully prescribed. A classical example is evaluation order where several languages allow an implementation to evaluate arguments in arbitrary order. As an example consider the following OCaml program:

```
((fun x -> fun y -> ()) (print_int 0)) (print_int 5)
```

With the left-to-right evaluation of the current native code backend, the inner `print_int 0` is evaluated first and we observe `05` being printed. With the right-to-left evaluation of the current bytecode backend, the outer-most `print_int 5` is evaluated first and we observe `50` being printed. With such non-determinism it is no longer immediate to check that the language implementation behaves as desired. Even though the behaviour of each individual implementation may be deterministic it is hard to judge two outputs equivalent up to the loose definition. This motivates extending the approach of Pałka et al. [2011] to generate type correct programs free of evaluation-order dependence. To this end we

- develop a novel type and effect system with an effect that indicates when the evaluation order is inconsequential for the observable behaviour of the program,
- develop a goal-directed, bottom-up generation algorithm that prevents us from generating programs with evaluation-order dependence,
- discuss our implementation of the approach including a shrinker of effectful programs, and
- discuss a number of errors we have discovered in the testing process.

For the rest of this paper we develop a type and effect system for evaluation-order dependence (Sec. 2), prove that the type and effect system is sound and that it anticipates effects as desired (Sec. 3), recall relevant background material on QuickCheck and goal-directed program generation (Sec. 4), present our type-and-effect-directed algorithm for program generation (Sec. 5), discuss our implementation of the approach (Sec. 6) and a number of errors we have found while testing OCaml's bytecode and native code backends against each other (Sec. 7), and finally compare and contrast to previous contributions within the area (Sec. 8).

2 TYPES AND EFFECTS FOR EVALUATION-ORDER DEPENDENCE

2.1 Syntax

Before developing the type and effect system we need to settle on a programming language. For presentational purposes we consider the following language, a minimal extension of the simply-typed lambda calculus.

$$
\begin{array}{rcll}
bt & ::= & \texttt{unit} \mid \texttt{int} \mid \ldots & \text{(base types)} \\
\tau & ::= & bt \mid \tau_1 \to \tau_2 & \text{(types)} \\
c & ::= & \texttt{()} \mid \texttt{i} \mid \ldots & \text{(constants)} \\
e & ::= & c \mid \texttt{x} \mid \texttt{fun x -> } e \mid e_0\ e_1 & \text{(expressions)} \\
\Gamma & ::= & \cdot \mid \Gamma, (\texttt{x} : \tau) & \text{(type environments)}
\end{array}
$$

$$\frac{\Delta(c) = \tau}{\Delta; \Gamma \vdash c : \tau}\ (\textsc{Const}) \qquad \frac{(\mathsf{x} : \tau) \in \Gamma}{\Delta; \Gamma \vdash \mathsf{x} : \tau}\ (\textsc{Var}) \qquad \frac{\Delta; \Gamma, (\mathsf{x} : \tau_1) \vdash e : \tau_2}{\Delta; \Gamma \vdash \mathsf{fun\ x\ ->}\ e : \tau_1 \longrightarrow \tau_2}\ (\textsc{Lam})$$

$$\frac{\Delta; \Gamma \vdash e_0 : \tau_1 \longrightarrow \tau_2 \qquad \Delta; \Gamma \vdash e_1 : \tau_1}{\Delta; \Gamma \vdash e_0\ e_1 : \tau_2}\ (\textsc{App})$$

Fig. 1. Typing rules for simply-typed lambda calculus

$$\frac{e_0 \xrightarrow{\eta} e_0'}{e_0\ e_1 \xrightarrow{\eta} e_0'\ e_1}\ (\textsc{AppL}) \qquad \frac{e_1 \xrightarrow{\eta} e_1'}{e_0\ e_1 \xrightarrow{\eta} e_0\ e_1'}\ (\textsc{AppR}) \qquad \frac{}{(\mathsf{fun\ x\ ->}\ e)\ val \xrightarrow{\epsilon} e[\mathsf{x} \mapsto val]}\ (\textsc{AppLam})$$

$$\frac{\delta(c\ val_1\ \ldots\ val_n) = (val, \eta)}{c\ val_1\ \ldots\ val_n \xrightarrow{\eta} val}\ (\textsc{AppDelta})$$

Fig. 2. Operational semantics with effect annotations

As types we include an unspecified number of base types (incl. `unit` and `int`) as well as function types. Similarly we include an unspecified number of constants (incl. `()` and integer constants `i`). We parameterize the typing relation over the typing of constants by a constant environment Δ that maps each constant c to its type. With this parameterization in mind the judgements are of the form $\Delta; \Gamma \vdash e : \tau$. We recall the typing rules for this language in Fig. 1.

2.2 Semantics

Next we define the semantics of our language with a small-step operational semantics in Fig. 2. Our operational semantics is non-deterministic to capture that the sub-expressions of an application may be evaluated in any order (left-to-right, right-to-left, interleaved, ...). Formally we capture the individual run-time effect of a computation step with an annotation on the corresponding transition step. We assume that run-time effects are given by some effect alphabet Σ. For example, for '*print i*', the effect of printing an integer i, we have $print\ i \in \Sigma$. We let ϵ denote 'no run-time effect'. Following standard practice within formal languages [Martin 1997], ϵ doubles as the empty string of run-time effects ($\epsilon \in \Sigma^*$). We furthermore let η range over $\Sigma \cup \{\epsilon\}$, and we let val range over the syntactic values of our language. These include function values as well as partially applied builtin primitives:

$$\begin{array}{lll} val\ ::= \mathsf{fun\ x\ ->}\ e & \text{where } fv(e) \subseteq \{\mathsf{x}\} & \text{(fun. values)} \\ \quad \mid c\ val_1\ \ldots\ val_{n-1} & \text{where } n \leq arity(c) & \text{(partially app. prim.)} \end{array}$$

where fv is defined as traditional:

$$fv(c) = \emptyset \qquad fv(\mathsf{fun\ x\ ->}\ e) = fv(e) \setminus \{\mathsf{x}\}$$
$$fv(\mathsf{x}) = \{\mathsf{x}\} \qquad fv(e_0\ e_1) = fv(e_0) \cup fv(e_1)$$

and where we assume some function $arity(c) : \mathbb{N} \cup \{0\}$ that associates to each constant its expected number of arguments. We assume that $arity$'s output is compatible with the constant's type signature $arity(c) = numargs(\Delta(c))$ where $numargs(bt) = 0$ and $numargs(\tau_0 \longrightarrow \tau_1) = 1 + numargs(\tau_1)$.

It is immediate from the above definitions that $fv(val) = \emptyset$ and that syntactic values cannot be reduced further. Substitution is (partially) defined as traditional:

$$c[\mathtt{y} \mapsto e] = c \qquad (\mathtt{fun\ x\ ->}\ e')[\mathtt{y} \mapsto e] = \begin{cases} \mathtt{fun\ x\ ->}\ e' & \mathtt{x} = \mathtt{y} \\ \mathtt{fun\ x\ ->}\ (e'[\mathtt{y} \mapsto e]) & \mathtt{x} \neq \mathtt{y} \wedge \mathtt{x} \notin fv(e) \end{cases}$$

$$\mathtt{x}[\mathtt{y} \mapsto e] = \begin{cases} e & \mathtt{x} = \mathtt{y} \\ \mathtt{x} & \mathtt{x} \neq \mathtt{y} \end{cases} \qquad (e_0\ e_1)[\mathtt{y} \mapsto e] = (e_0[\mathtt{y} \mapsto e])\ (e_1[\mathtt{y} \mapsto e])$$

By the bound variable convention [Pierce 2002] for the lambda case we can always α-rename the bound variable so as to avoid variable capture. For this reason we will also identify expressions up to α-renaming.

As constants we include `()` such that $\Delta(\mathtt{()}) = \mathtt{unit}$ and *print_int* such that $\Delta(\mathit{print\_int}) = \mathtt{int} \longrightarrow \mathtt{unit}$. For example, for `()` we have $\mathit{arity}(\mathtt{()}) = \mathit{numargs}(\mathtt{unit}) = 0$ and for *print_int* we have $\mathit{arity}(\mathit{print\_int}) = \mathit{numargs}(\mathtt{int} \longrightarrow \mathtt{unit}) = 1$. Note how we distinguish between `print_int` (a variable in verbatim font, bound in an initial environment Γ) and *print_int* (a primitive or constant in italic font, bound in Δ), with the latter typically bound to the former.

2.3 The Type and Effect System

Expressions are evaluated by value (in applicative order), but our observations may depend on whether this order is left-to-right or right-to-left. The basic idea is therefore to track the evaluation-order dependence with two bits:

- with one bit we track whether evaluation of an expression (or function) may have an effect and
- with another bit we track whether the observable outcome of the evaluation of an expression (or function) may depend on the evaluation order.

To this end we extend the grammar of types to the following grammar for the type and effect system:

$$\begin{array}{lll} \tau & ::= & bt \mid \tau_1 \xrightarrow{\varphi} \tau_2 \qquad \text{(effect types)} \\ \varphi & ::= & ef/ev \quad \text{where } ef, ev \in \{\mathtt{tt}, \mathtt{ff}\} \qquad \text{(effects)} \\ \Gamma & ::= & \cdot \mid \Gamma, (\mathtt{x} : \tau) \qquad \text{(type and effect environments)} \end{array}$$

Overall our type and effect system judgements are now of the form: $\Delta, \Gamma \vdash e : \tau \,\&\, \varphi$ where τ ranges over effect types and φ ranges over a pair of effect bits ef/ev where ef and ev are either `tt` or `ff`. The constant environment Δ and the typing environment Γ are similarly adjusted to map constants and variables to effect types. For example, $\Delta(\mathit{print\_int}) = \mathtt{int} \xrightarrow{\mathtt{tt/ff}} \mathtt{unit}$. Finally the *numargs* function is similarly adjusted to count top-level arrows in effect types. We assume that effects of primitives are limited to full applications, such that they will all be on the form $\Delta(c) = \tau_1 \xrightarrow{\mathtt{ff/ff}} \ldots \xrightarrow{\mathtt{ff/ff}} \tau_n \xrightarrow{ef/\mathtt{ff}} \tau$ where $n = \mathit{arity}(c)$. The resulting type and effect system is presented in Fig. 3. As traditional [Pierce 2002] a system with a sub-typing relation can either be formulated in a *declarative* manner or in an *algorithmic* manner, depending on whether the sub-typing relation can be applied unrestrictedly or whether it is spliced into the syntax-directed rules. For presentational purposes our formulation in Fig. 3 is algorithmic since we will later use the system to drive a generator of type-and-effect correct terms. In particular the system allows sub-typing in the leaves (EConst) and (EVar) and sub-effecting in all four rules (note how φ is a free variable in the conclusion of (EConst), (EVar), and (ELam)).

Our effects are naturally ordered by a componentwise partial ordering defined by rule (EOrder) in Fig. 4. The effect ordering expresses that a less effectful computation with effect `ff/ff` is considered

$$\frac{\Delta(c) = \tau \qquad \tau \sqsubseteq \tau'}{\Delta; \Gamma \vdash c : \tau' \,\&\, \varphi}\ (\textsc{EConst}) \qquad \frac{(\mathsf{x} : \tau) \in \Gamma \qquad \tau \sqsubseteq \tau'}{\Delta; \Gamma \vdash \mathsf{x} : \tau' \,\&\, \varphi}\ (\textsc{EVar})$$

$$\frac{\Delta; \Gamma, (\mathsf{x} : \tau_1) \vdash e : \tau_2 \,\&\, \varphi_1}{\Delta; \Gamma \vdash \mathsf{fun\ x\ ->}\ e : \tau_1 \xrightarrow{\varphi_1} \tau_2 \,\&\, \varphi}\ (\textsc{ELam})$$

$$\frac{\Delta; \Gamma \vdash e_0 : \tau_1 \xrightarrow{\varphi} \tau_2 \,\&\, \varphi_0 \qquad \Delta; \Gamma \vdash e_1 : \tau_1 \,\&\, \varphi_1 \qquad \varphi \sqcup \varphi_0 \sqcup \varphi_1 \sqsubseteq \varphi' \qquad \varphi_0 \sqcap \varphi_1 \Rrightarrow \varphi'}{\Delta; \Gamma \vdash e_0\ e_1 : \tau_2 \,\&\, \varphi'}\ (\textsc{EApp})$$

Fig. 3. Effect type system

$$\frac{ef \Longrightarrow ef' \qquad ev \Longrightarrow ev'}{ef/ev \sqsubseteq ef'/ev'}\ (\textsc{EOrder}) \qquad \frac{}{bt \sqsubseteq bt}\ (\textsc{BTReflSub})$$

$$\frac{\tau_0' \sqsubseteq \tau_0 \qquad \varphi \sqsubseteq \varphi' \qquad \tau_1 \sqsubseteq \tau_1'}{\tau_0 \xrightarrow{\varphi} \tau_1 \sqsubseteq \tau_0' \xrightarrow{\varphi'} \tau_1'}\ (\textsc{FunSub})$$

Fig. 4. Effect ordering, sub-typing, and type-and-effect ordering

less than a more effectful computation with effect `tt/ff`. This ordering inherits reflexivity and transitivity by the underlying implication ordering. In addition to the effect ordering we include a sub-typing ordering in Fig. 4. In this sub-type ordering base types are ordered and function types are ordered contra-variantly in the argument and co-variantly in the effect and result types. Overall the sub-typing ordering allows us to use a less effectful computation in place of a more effectful one.

Finally the rule (EApp) in Fig. 3 utilizes an implication operator $\Rrightarrow$ over effect pairs. It is defined as: $\mathtt{tt}/ev \Rrightarrow \mathtt{tt}/\mathtt{tt}$ and $\mathtt{ff}/ev \Rrightarrow ef'/ev'$. This constraint expresses that if both operator and operand of an application may have effects then the observable outcome may depend on the evaluation order. We observe that $\Rrightarrow$ is monotone in the conclusion.

As an example, consider ((`fun x -> fun y -> ()`) (`print_int 0`)) (`print_int 5`) from the introduction in an initial type environment containing `print_int` : $\mathtt{int} \xrightarrow{\mathtt{tt/ff}} \mathtt{unit}$. By an application of the (EApp) rule we can easily build a sub-derivation tree for each `print_int` call (where we leave out the effect constraints for readability):

$$\frac{(\mathtt{print\_int} : \mathtt{int} \xrightarrow{\mathtt{tt/ff}} \mathtt{unit}) \in \Gamma}{\Delta; \Gamma \vdash \mathtt{print\_int} : \mathtt{int} \xrightarrow{\mathtt{tt/ff}} \mathtt{unit} \,\&\, \mathtt{ff/ff}}\ (\textsc{EVar}) \qquad \frac{\Delta(\mathtt{0}) = \mathtt{int}}{\Delta; \Gamma \vdash \mathtt{0} : \mathtt{int} \,\&\, \mathtt{ff/ff}}\ (\textsc{EConst})$$

$$\frac{}{\Delta; \Gamma \vdash \mathtt{print\_int\ 0} : \mathtt{unit} \,\&\, \mathtt{tt/ff}}\ (\textsc{EApp})$$

With this sub-tree in mind we can construct a derivation tree for the entire example program as illustrated in Fig. 5. The derivation tree illustrates how the type and effect system correctly detects that evaluation of the example may have an effect (the first `tt` in the conclusion) and that the outcome may depend on the evaluation order (the second `tt` in the conclusion).

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{}{\Delta(\,()\,) = \texttt{unit}}}{\Delta;\Gamma,(\texttt{x}:\texttt{unit}),(\texttt{y}:\texttt{unit}) \vdash \,()\, : \texttt{unit}\,\&\,\texttt{ff/ff}}\,(\textsc{EConst})}{\Delta;\Gamma,(\texttt{x}:\texttt{unit}) \vdash \texttt{fun y -> }\,()\, : \texttt{unit} \xrightarrow{\texttt{ff/ff}} \texttt{unit}\,\&\,\texttt{ff/ff}}\,(\textsc{ELam})}{\Delta;\Gamma \vdash \texttt{fun x -> fun y -> }\,()\, : \texttt{unit} \xrightarrow{\texttt{ff/ff}} \texttt{unit} \xrightarrow{\texttt{ff/ff}} \texttt{unit}\,\&\,\texttt{ff/ff}}\,(\textsc{ELam})}{\dfrac{\Delta;\Gamma \vdash \texttt{print\_int 0} : \texttt{unit}\,\&\,\texttt{tt/ff}}{\Delta;\Gamma \vdash (\texttt{fun x -> fun y -> }\,()\,)\,(\texttt{print\_int 0}) : \texttt{unit} \xrightarrow{\texttt{ff/ff}} \texttt{unit}\,\&\,\texttt{tt/ff}}\,(\textsc{EApp})}{\dfrac{\Delta;\Gamma \vdash \texttt{print\_int 5} : \texttt{unit}\,\&\,\texttt{tt/ff}}{\Delta;\Gamma \vdash ((\texttt{fun x -> fun y -> }\,()\,)\,(\texttt{print\_int 0}))\,(\texttt{print\_int 5}) : \texttt{unit}\,\&\,\texttt{tt/tt}}\,(\textsc{EApp})}$$

Fig. 5. Example derivation tree for the program ((fun x -> fun y -> ()) (print_int 0)) (print_int 5). For brevity the effect constraints have been omitted and so has the sub-trees for both print_int-calls

A change of representation. If the observable outcome of evaluation of an expression depends on the evaluation order the evaluation must also have an effect. As such we only need the *reduced product domain* [Cousot and Cousot 1977] with three effect values: $\bot \sqsubseteq \mathit{eff} \sqsubseteq \mathit{evalorder}$ where $\bot$ denotes 'the program has no effect' (ff/ff), *eff* denotes 'the program may have an effect' (tt/ff), and *evalorder* denotes 'the observable output may depend on evaluation order' (tt/tt). For presentational reasons we choose to stick with the more intuitive two-bit representation in the rest of this paper.

3 SOUNDNESS OF TYPE AND EFFECT SYSTEM

Before we proceed with building a type-and-effect-guided generator, we prove soundness of the type and effect system. To keep our formalization manageable we limit ourselves to effects expressible as transition annotations in our semantics. This includes printing, writing to files, etc. We first characterize the effects of syntactic values.

Lemma 3.1 (Typing of syntactic values). *If* $\Delta;\Gamma \vdash \mathit{val} : \tau\,\&\,\varphi$ *then* $\Delta;\Gamma \vdash \mathit{val} : \tau\,\&\,\varphi'$

In particular we may choose $\varphi' = \texttt{ff/ff}$ meaning that syntactic values have no effects. The proof follows by a straight-forward structural induction on *val*. Secondly we prove a standard result [Pierce 2002] characterizing the syntactic shape of particular types.

Lemma 3.2 (Canonical forms). *If* $\Delta;\Gamma \vdash \mathit{val} : \tau\,\&\,\varphi$ *then*

- $\mathit{val} = c\ \mathit{val}_1 \ldots \mathit{val}_{n-1}$ *for some* $n \le \mathit{arity}(c)$ *and* $\Delta;\Gamma \vdash c : \tau_1 \xrightarrow{\texttt{ff/ff}} \cdots \xrightarrow{\texttt{ff/ff}} \tau_{n-1} \xrightarrow{\texttt{ff/ff}} \tau\,\&\,\texttt{ff/ff}$ *with* $\Delta;\Gamma \vdash \mathit{val}_1 : \tau_1\,\&\,\texttt{ff/ff}, \ldots, \Delta;\Gamma \vdash \mathit{val}_{n-1} : \tau_{n-1}\,\&\,\texttt{ff/ff}$
- *or* $\tau = \tau_0 \xrightarrow{\varphi'} \tau_1$ *and* $\mathit{val} = \texttt{fun x -> } e$

The proof follows by induction on the typing derivation. Similarly we can relax the type in the typing environment. Again this follows by induction over the typing derivation and with appeal to the bound variable convention [Pierce 2002].

Lemma 3.3 (Type environment relaxation).
If $\Delta;\Gamma,(\texttt{x}:\tau'),\Gamma' \vdash e : \tau\,\&\,\varphi$ *and* $\tau'' \sqsubseteq \tau'$ *then* $\Delta;\Gamma,(\texttt{x}:\tau''),\Gamma' \vdash e : \tau\,\&\,\varphi$

The following lemma lets us replace the type and effect in a derivation with a greater type and effect, akin to a separate sub-typing rule in a declaratively formulated system [Pierce 2002]. The proof follows by induction on the typing derivation, by transitivity of the sub-typing relation, by appeal to Lemma 3.3, and by monotonicity of the implication operation $\Rrightarrow$ over effects.

Lemma 3.4 (Sub-typing and sub-effecting).
If $\Delta; \Gamma \vdash e : \tau \,\&\, \varphi$ *and* $\tau \sqsubseteq \tau'$ *and* $\varphi \sqsubseteq \varphi'$ *then* $\Delta; \Gamma \vdash e : \tau' \,\&\, \varphi'$

Finally we will need a traditional substitution lemma to help with proving preservation.

Lemma 3.5 (Substitution lemma).
If $\Delta; \Gamma, (\mathsf{x} : \tau') \vdash e : \tau \,\&\, \varphi$ *and* $\Delta; \Gamma \vdash e' : \tau' \,\&\, \mathtt{ff}/\mathtt{ff}$ *and* $\mathsf{x} \notin fv(e')$ *then* $\Delta; \Gamma \vdash e[\mathsf{x} \mapsto e'] : \tau \,\&\, \varphi$

The substitution lemma lets us replace a variable x with a type-and-effect correct expression e' of the same type (albeit without effects). The lemma follows by structural induction on e and by appeal to Lemma 3.4 and the bound variable convention.

3.1 Preservation and Progress

We are now in position to prove a traditional preservation and progress result for the type and effect system [Wright and Felleisen 1994]. We assume that $\delta(c\ val_1\ \ldots\ val_n)$ is defined only when the builtin primitive is fully applied to $n = arity(c) \geq 1$ arguments of the right type: $\Delta(c) = \tau_1 \xrightarrow{\mathtt{ff}/\mathtt{ff}} \ldots \xrightarrow{\mathtt{ff}/\mathtt{ff}} \tau_n \xrightarrow{ef/\mathtt{ff}} \tau$ with $\Delta; \cdot \vdash val_1 : \tau_1 \,\&\, \mathtt{ff}/\mathtt{ff}, \ldots \Delta; \cdot \vdash val_n : \tau_n \,\&\, \mathtt{ff}/\mathtt{ff}$ and such that the resulting value and the effect of the primitive is soundly accounted for: if $\delta(c\ val_1\ \ldots\ val_n) = (val, \eta)$ then $\Delta; \cdot \vdash val : \tau \,\&\, \mathtt{ff}/\mathtt{ff}$ and if $\eta \neq \epsilon$ then $ef = \mathtt{tt}$.

Theorem 3.6 (Preservation and progress).

- *if* $\Delta; \cdot \vdash e : \tau \,\&\, \varphi$ *and there exists* e' *such that* $e \xrightarrow{\eta} e'$ *then* $\Delta; \cdot \vdash e' : \tau \,\&\, \varphi$
- *if* $\Delta; \cdot \vdash e : \tau \,\&\, \varphi$ *then either* $e = val$ *or there exists* e' *such that* $e \xrightarrow{\eta} e'$

The preservation part follows by induction on the derivation for $\Delta; \cdot \vdash e : \tau \,\&\, \varphi$ and with appeal to Lemmas 3.1, 3.4, and 3.5, and our assumption about the soundness of fully applied primitives. The progress part follows by structural induction on e and by appeal to Lemmas 3.1 and 3.2.

3.2 Soundness of the Effect Bit

The preservation and progress result captures soundness of the underlying type system but it says nothing about the soundness of the effect bits. Secondly we therefore characterize formally the meaning of the two bits. We do so one bit at a time. In formalizing the first effect bit we initially express one step soundness and then lift this to soundness over traces.

Lemma 3.7 (One-step soundness of effect bit).
If $\Delta; \Gamma \vdash e : \tau \,\&\, ef/ev$ *and* $e \xrightarrow{\eta} e'$ *and* $\eta \neq \epsilon$ *then* $ef = \mathtt{tt}$

The proof follows by structural induction on e and by appeal to the soundness of the effects of fully applied primitives. We can now lift the one-step result to traces. The proof proceeds by induction on n and by appeal to Lemma 3.7 and preservation (Theorem 3.6). In words this first theorem expresses that if a type-and-effect correct program e has an effect observable during its evaluation then the effect is correctly anticipated by the type and effect system.

Theorem 3.8 (Soundness of effect bit).
If $\Delta; \Gamma \vdash e : \tau \,\&\, ef/ev$ *and* $e \xrightarrow{\eta_1} e_1 \xrightarrow{\eta_2} \ldots \xrightarrow{\eta_n} e_n$ *and there exists* i *such that* $\eta_i \neq \epsilon$ *then* $ef = \mathtt{tt}$

3.3 Soundness of the Evaluation-Order Bit

We finally prove soundness of the second evaluation-order bit. In words this second theorem expresses that if the observable behaviour of a type-and-effect correct program e depends on the evaluation order then it is correctly anticipated by the type and effect system. This is most easily

seen from the contrapositive statement: if from some type-and-effect correct expression there exists two different traces of observable effects $\eta_1\eta_2\ldots\eta_n \neq \eta'_1\eta'_2\ldots\eta'_{n'}$ then $ev = \mathtt{tt}$.

Theorem 3.9 (Soundness of evaluation-order bit).
If $\Delta;\Gamma \vdash e : \tau \,\&\, ef/ev$ *and* $e \xrightarrow{\eta_1} e_1 \xrightarrow{\eta_2} \ldots \xrightarrow{\eta_n} e_n = val$ *and* $e \xrightarrow{\eta'_1} e'_1 \xrightarrow{\eta'_2} \ldots \xrightarrow{\eta'_{n'}} e'_{n'} = val'$ *and* $ev = \mathtt{ff}$ *then* $\eta_1\eta_2\ldots\eta_n = \eta'_1\eta'_2\ldots\eta'_{n'}$

In order to prove this property we need a couple of helper lemmas. First of all we need a diamond property to connect two multi-step traces modulo the effects on the transition relations. We first prove a one-step version:

Lemma 3.10 (One-step diamond property up to effects).
If $e \xrightarrow{\eta_l} e_l$ *and* $e \xrightarrow{\eta_r} e_r$ *then either* $e_l = e_r$ *or there exists* e' *such that* $e_l \xrightarrow{\eta'_l} e'$ *and* $e_r \xrightarrow{\eta'_r} e'$

The proof follows by structural induction on e. We can subsequently lift this lemma to a multi-step diamond property. There are several approaches in the literature to such a proof. We choose a syntactic proof by induction in the pair of lengths (n, n') ordered lexicographically (a known well-ordering).

Lemma 3.11 (Multi-step diamond property up to effects).
If $e \xrightarrow{\eta_1} e_1 \xrightarrow{\eta_2} \ldots \xrightarrow{\eta_n} e_n$ *and* $e \xrightarrow{\eta'_1} e'_1 \xrightarrow{\eta'_2} \ldots \xrightarrow{\eta'_{n'}} e'_{n'}$ *then there exists traces* $e_n \xrightarrow{\eta_{n+1}} \ldots \xrightarrow{\eta_{n+n'}} e_{n+n'}$ *and* $e'_{n'} \xrightarrow{\eta'_{n'+1}} \ldots \xrightarrow{\eta'_{n'+n}} e'_{n'+n}$ *such that* $e_{n+n'} = e'_{n'+n}$

Secondly we need to realize that run-time effects are deterministic: we cannot take two steps with different run-time effects to the same target expression. Again this follows by structural induction on e.

Lemma 3.12 (Determinism of run-time effects). *If* $e \xrightarrow{\eta} e'$ *and* $e \xrightarrow{\eta'} e'$ *then* $\eta = \eta'$

Thirdly we need a one-step diamond property for expressions deemed evaluation-order independent. This lemma allows us to complete a diamond shape such that the effects along each side agrees. It follows by structural induction on e and by appeal to Lemma 3.7.

Lemma 3.13 (One-step diamond property of run-time effects). *If* $\Delta;\Gamma \vdash e : \tau \,\&\, ef/\mathtt{ff}$ *and* $e \xrightarrow{\eta_l} e_l$ *and* $e \xrightarrow{\eta_r} e_r$ *and* $e_l \neq e_r$ *then there exists* e' *such that and* $e_l \xrightarrow{\eta'_l} e'$ *and* $e_r \xrightarrow{\eta'_r} e'$ *and* $\eta_l\eta'_l = \eta_r\eta'_r$

Finally we are in position to prove the work-horse lemma of our desired theorem. The proof proceeds by induction on the length of the shortest of the two traces and by appeal to Lemma 3.11, 3.12, and 3.13, and preservation (Theorem 3.6).

Lemma 3.14 (Multi-step diamond property with run-time effects). *If* $\Delta;\Gamma \vdash e : \tau \,\&\, ef/ev$ *and* $e \xrightarrow{\eta_1} e_1 \xrightarrow{\eta_2} \ldots \xrightarrow{\eta_n} e_n$ *and* $e \xrightarrow{\eta'_1} e'_1 \xrightarrow{\eta'_2} \ldots \xrightarrow{\eta'_{n'}} e'_{n'}$ *and* $e_n = e'_{n'} = val$ *and* $ev = \mathtt{ff}$ *then* $\eta_1\eta_2\ldots\eta_n = \eta'_1\eta'_2\ldots\eta'_{n'}$ *and* $n = n'$.

From this lemma it is now straightforward to prove Theorem 3.9: By Lemma 3.11 it follows that $val = val'$ and from Lemma 3.14 the theorem now follows.

We remark that a constant effect annotation $\Gamma \vdash e : \tau \,\&\, \mathtt{tt}/\mathtt{tt}$ will satisfy this soundness criterion, corresponding to the pessimistic prediction that any program *may have* an effect and *may be* evaluation-order dependent. In general we seek the *least* type and effect for a given program.

4 TYPE-DRIVEN PROGRAM GENERATION: STATE OF THE ART

With a sound effect system in place we now return to the primary topic, namely testing. We first recall the basics of QuickCheck [Claessen and Hughes 2000] as well as the goal-directed program generation approach of Pałka et al. [2011]. For the rest of this article we will use OCaml and the QCheck library[1] for QuickChecking but the approach could just as easily have been carried out, e.g., in Haskell or F#.

4.1 QuickCheck

QuickCheck builds on the idea of substituting hand-written tests with *randomized property-based tests*. A family of tests are instead expressed by *a property*, e.g., the list property $\forall l, l'.$ `rev` $(l \,@\, l') =$ (`rev` l') @(`rev` l) and *a generator* supplying values to fill in for the quantified list variables l and l'. Each instance of the property thereby gives rise to a test case:

$$\texttt{rev}\ ([1;2;4]\,@[3]) = (\texttt{rev}\ [3])\,@(\texttt{rev}\ [1;2;4]),$$
$$\texttt{rev}\ ([]\,@[9]) = (\texttt{rev}\ [9])\,@(\texttt{rev}\ []),$$
$$\texttt{rev}\ ([7]\,@[5;8]) = (\texttt{rev}\ [5;8])\,@(\texttt{rev}\ [7]),\ \ldots$$

Functional programming is well suited both for describing properties and for building generators. For example, the property above can easily be expressed by the Boolean-valued function:

```
(fun (l,l') -> List.rev (l @ l') = (List.rev l') @ (List.rev l))
```

Similarly, generators expressed as combinators compose easily. For example, we can pass a builtin generator of integers `small_int` as an argument to the parametric list generator to build a generator of integer lists: `list small_int`. Similarly the parametric pair generator accepts generators for each of their components: `pair (list small_int) (list small_int)`. All in all, we can thereby describe the above QuickCheck test as follows (where we supply a test title string as an optional argument):

```
let rev_concat_test =
  Test.make ~name:"reverse-concat"
    (pair (list small_int) (list small_int))
    (fun (l,l') -> List.rev (l @ l') = (List.rev l') @ (List.rev l))
```

and run it with `QCheck_runner.run_tests_main [rev_concat_test]` which will (by default) check 100 instances:

```
random seed: 171662158
law reverse-concat: 100 relevant cases (100 total)
success (ran 1 tests)
```

When a property fails to hold it is essential to present a counterexample to the developer. Such a failure may be caused by an error in the specification or in the code under test. For example, suppose we test the following erroneous specification (note how `l` and `l'` are switched):

```
(fun (l,l') -> List.rev (l @ l') = (List.rev l) @ (List.rev l'))
```

the QuickCheck framework will present a counterexample:

```
law buggy reverse-concat: 1 relevant cases (1 total)
  test `buggy reverse-concat`
  failed on ≥ 1 cases:
  ([0], [1]) (after 362 shrink steps)
```

[1] https://github.com/c-cube/qcheck/

From this minimal counterexample it should be clear that the above property does not hold: The left-hand-side `List.rev ([0]@[1])` yields `[1;0]` whereas `(List.rev [0])@(List.rev [1])` on the right-hand-side yields `[0;1]`. The counterexample has however been cut down (*shrunk*) in a post-processing phase in order to make it easier for humans to comprehend. If we disable the shrinker the tester may for example be presented with the following counterexample instead:

```
law buggy reverse-concat: 1 relevant cases (1 total)
  test `buggy reverse-concat`
  failed on ≥ 1 cases:
  ([84; 9; 9; 467; 82; 58; 3; 78; 1; 2; 1390; 52; 16; 3; 153; 4; 3; 0;
6; 18; 2; 637; 48; 1; 878; 0; 2; 19; 9836; 12; 1; 89; 8; 8; 3; 0; 8; 0;
6; 1], [6862; 41; 9; 8; 44; 5; 80; 4; 276])
```

It is less clear from this bigger counterexample what is causing the failure.

4.2 Generating Type-Correct Programs

To test the backend of a language implementation we need input that makes it through the frontend, and in particular through the strict guarding of a static type checker. One can generate such programs following an approach of Pałka et al. [2011].

The basic idea is to generate programs following the rules of the type system, albeit by reading the typing relation in a goal-directed, bottom-up manner. Suppose we wish to generate an expression of type `int` as illustrated in Fig. 6. We can do so by matching the `int` type against the conclusion of the rules in Fig. 1 and realize that we are able to do so by either the (Const), (Var), or (App) rules. Suppose we arbitrarily choose the (App) rule. To successfully apply this rule we must come up with an argument type τ_1. A type generator can produce such arbitrary types, so suppose it returns `int`. We now recursively invoke the expression generator twice to generate sub-expressions of type $\tau_1 \to \texttt{int} = \texttt{int} \to \texttt{int}$ and $\tau_1 = \texttt{int}$. To produce an expression of function type we may for example choose the (Lam) rule. This will in turn generate an arbitrary variable name `x`, add the binding `(x : int)` to the type environment, and recursively invoke the expression generator to generate a sub-expression of type `int` in this extended type environment. In this recursive call we may choose the (Var) rule to generate an integer expression in the extended type environment. In particular the variable `x` has type `int` and thereby satisfies our goal. The second recursive call from the (App) rule with the goal of generating an expression with `int` may be fulfilled by the (Const) rule by returning any constant, e.g., `42`.

From a generation point of view, to generate a call to, e.g., the function `(+) : int → int → int` for curried addition of integers we need to choose the (App) rule twice, in both cases choose `int` as the arbitrary argument type, and only in the innermost case choose the (Var) rule and the `(+)` operation bound in Γ. Furthermore, repeated applications of the (App) rules may send the generator off on searches for functions of greater and greater types. As a remedy Pałka et al. [2011] therefore suggest to add the following (Indir) rule:

$$\frac{(\mathtt{f} : \tau_1 \to \ldots \to \tau_n \to \tau) \in \Gamma \qquad \Delta;\Gamma \vdash e_1 : \tau_1 \qquad \ldots \qquad \Delta;\Gamma \vdash e_n : \tau_n}{\Delta;\Gamma \vdash \mathtt{f}\ e_1 \ldots e_n : \tau} \text{ (Indir)}$$

Logically (Indir) is derivable from the remaining rules. However from a generation point of view the (Indir) rule instead expresses environment-driven applications: it expresses choosing a known receiver in scope with the desired result type τ and only invoke the expression generator recursively to produce arguments of the appropriate types $\tau_1, \ldots, \tau_n$, thereby providing a more guided search for well-typed programs.

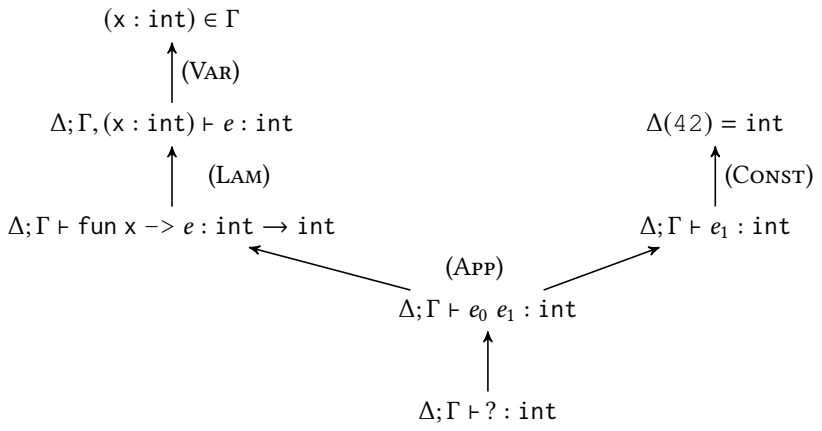


Fig. 6. Goal-directed generation of the program (fun x -> x) 42 : int

One can extend this approach to also support polymorphic functions [Pałka et al. 2011]. As an example, consider List.hd : α list → α. If our goal is to generate an expression of type int, List.hd is a valid receiver in (Indir) because α can be instantiated to the concrete type int. If we choose to do so, we need to recursively generate an argument with the same instantiation α list[$\alpha \mapsto$ int] = int list. In general, a polymorphic function can be applied if one of the receiver's result types can be unified with the goal type. We therefore extend types to include type variables:

$$\tau ::= \ldots \mid \alpha \qquad \text{(types)}$$

In our view this approach does not suffice for testing the OCaml compiler backends against one another. The reason is that the evaluation order of OCaml, like, e.g., C and Scheme, is loosely defined. The bytecode interpreter, following the original ZINC bytecode interpreter, evaluates applications right-to-left to save needless closure allocations [Leroy 1990] whereas OCaml's later native code backend may evaluate left-to-right. In fact a direct implementation of the approach of Pałka et al. [2011], as we did in an unpublished report [Kasting and Justesen 2016], produces too many counterexamples that merely illustrate observable differences caused by different evaluation orders. This motivates the development of a refined approach that only generates evaluation-order independent programs.

5 EFFECT-DRIVEN PROGRAM GENERATION

In order to avoid generating programs with evaluation-order dependence, we refine the approach of Pałka et al. [2011] to instead generate programs driven by our type and effect system. Specifically we utilize the type and effect rules described in Sec. 2 in order to generate well-typed programs with effect tt/ff, thereby generating effectful but evaluation-order independent programs. This may in turn involve invoking the generator recursively to generate sub-expressions with and without effects.

As in the guided, type-driven approach we include the following derived rules, to guide our generator towards calls to known receivers in scope, similarly to the (Indir) rule:

$$\frac{(\mathtt{f} : \tau_1 \xrightarrow{\mathtt{ff/ff}} \ldots \xrightarrow{\mathtt{ff/ff}} \tau_n \xrightarrow{\mathtt{ff/ff}} \tau) \in \Gamma \qquad \Delta;\Gamma \vdash e_1 : \tau_1 \,\&\, \mathtt{ff/ff} \quad \ldots \quad \Delta;\Gamma \vdash e_n : \tau_n \,\&\, \mathtt{ff/ff}}{\Delta;\Gamma \vdash \mathtt{f}\ e_1 \ldots e_n : \tau \,\&\, \mathtt{ff/ff}}\ (\textsc{EIndir1})$$

$$\frac{\begin{array}{c}(\mathtt{f} : \tau_1 \xrightarrow{ef_1/\mathtt{ff}} \ldots \xrightarrow{ef_{n-1}/\mathtt{ff}} \tau_n \xrightarrow{ef_n/\mathtt{ff}} \tau) \in \Gamma \\ \Delta;\Gamma \vdash e_1 : \tau_1 \,\&\, ef'_1/\mathtt{ff} \quad \ldots \quad \Delta;\Gamma \vdash e_n : \tau_n \,\&\, ef'_n/\mathtt{ff} \\ i \in [1;n] \qquad \forall j < i.\ ef_j = \mathtt{ff} \qquad \forall j \neq i.\ ef'_j = \mathtt{ff} \qquad ef'_i = \mathtt{tt}\end{array}}{\Delta;\Gamma \vdash \mathtt{f}\ e_1 \ldots e_n : \tau \,\&\, \mathtt{tt/ff}}\ (\textsc{EIndir2})$$

A word of explanation is required for these rules' preconditions:

- For the rule EIndir1 it should be intuitive that in an effect-free application none of the arguments can have effects, nor can the receiver when supplied with the corresponding values.
- For the rule EIndir2 the reasoning is more complex. As a motivating example consider a receiver $(\mathtt{f} : \mathtt{unit} \xrightarrow{\mathtt{tt/ff}} \mathtt{unit} \xrightarrow{\mathtt{ff/ff}} \mathtt{unit}) \in \Gamma$ and the application `f ()` (`print_int 0`). When evaluated from right-to-left we first observe print 0 and subsequently observe the effect of `f ()`. On the other hand when evaluated from left-to-right we first observe the effect of `f ()` and subsequently observe print 0. To prevent such evaluation-order dependence in the derived rule we must therefore ensure that
  - at most one actual parameter e_i can have an effect and
  - any effects from partial applications can occur only from index i and upwards.

One can prove that these rules follow from rules (EVar) and (EApp) by induction in the number of arguments n: Any type-and-effect proof using (EIndir1) and (EIndir2) can be transformed into a proof using only (EVar) and (EApp). Our splitting of the indirection rule into two separate rules (Indir1) and (Indir2) is however only for presentational purposes: henceforth we consider these two cases as a collective (Indir) case.

At any point during the algorithm, we have a goal, which is the type and effect of the expression we want to generate. Based on this goal we assemble a list of applicable rules:

EConst is applicable if the goal type matches a constant's type. Since a constant does not have an effect in itself this rule is applicable regardless of the goal effect.

EVar is applicable if there is variable in the type environment with a type compatible with the goal. Since a variable does not have an effect in itself this rule is applicable regardless of the goal effect.

ELam is only applicable if the goal is a function type. Since a function literal does not have an effect in itself this rule is applicable regardless of the goal effect.

EApp is always applicable. However since the type τ_1 of the parameter is not determined by the goal we first generate a random type τ_1 before adding this rule to the list. We flip a coin to pass the goal effect to either the operator or the operand and pass `ff/ff` to the other. Furthermore we also pass the goal effect φ as the annotated effect in the type goal $\tau_1 \xrightarrow{\varphi} \tau_2$ to the operator call.

EIndir is applicable if there is a variable in the type environment such that the variable matches the goal. A variable matches the goal type if it's type can be instantiated to the goal

type when either fully applied, partially applied, or not applied at all. The effect signature of a chosen receiver can at most contain effects matching the desired goal effect. Finally we compute the index i of the left-most `tt/ff` effect annotation in the receivers signature (or the number of needed arguments if there are no effects), pass the goal effect along as the goal effect to exactly one of the arguments (chosen uniformly in $[1, \ldots, i]$), and pass `ff/ff` to the others.

After we have assembled the list of applicable rules, we randomly select one and recursively invoke the generator algorithm in order to generate the premises of the selected rule. If all recursive calls return successfully then we construct the term in accordance with the selected rule. If any of the recursive calls fails the probability of success on a retry is small, so we remove the rule from the list and randomly select one of the remaining rules for generation. If the list of applicable rules becomes empty, the generation has failed and we return an error value.

Without a base case the algorithm is not guaranteed to terminate as EApp is always applicable. This may trigger an infinite search for inhabitants of bigger and bigger types. We remedy the issue by imposing a size limit on the generated terms, as is standard within QuickChecking. This value is added as a parameter to the algorithm and decreased on the recursive calls. In the base case if the size parameter reaches zero we only attempt to apply the axioms (the (EVar) or (EConst) rules). By using a size parameter and keeping track of the rules that have already been used and failed, we ensure that the algorithm always terminates. To generate programs of type τ that are free of evaluation-order dependence, we initially invoke the generator algorithm with the goal τ & `tt/ff`.

As an alternative implementation strategy, one could express a generator of evaluation-order independent programs as a combination of two phases: a first phase would produce type correct programs following the approach of Pałka et al. [2011] and a subsequent phase would then filter away the programs that do not pass a type and effect check. One can view our current approach as an optimization of such a two-phase approach, in that we simply eliminate the option of generating candidate programs that would not make it through the filter.

6 IMPLEMENTATION

We have implemented the generation algorithm as described in Sec. 5. Type-wise our implementation supports unit, bool, int, string, list, and function types. In addition to the four expression constructs presented so far the implementation also supports the generation of let-bindings and conditionals according to the rules in Fig. 7. Following Fig. 3 these rules are also phrased with algorithmic sub-effecting. From a generation point of view the effect φ in the (ELet) and (EIf) rules is given as a parameter, which motivated the current formulation. Their formulation is not entirely arbitrary though. For example, based on standard desugaring of a let-binding into an immediate application, one can transform a type and effect proof using (ELet) into one using (EApp) and (ELam). Intuitively, a type and effect proof using (ELet) corresponds to an (ELet)-free proof in which we have applied sub-effecting (Lemma 3.4) to promote the effect of both e_1 and e_2 to an upper bound of them both.

As the initial size parameter the `sized` generators of QCheck chooses an arbitrary integer. This allows us to generate relatively large example programs. We seed the initial environment Γ with a number of bindings from OCaml's standard library, primarily from the (initially opened) `Pervasives` module. These include operations with effects that go beyond our formalized fragment. For example, we include `List.hd` that may throw a `Failure` exception when applied to an empty list, curried integer division `(/)` that may throw a `Division_by_zero` exception, and `print_int` from the introduction.

The backtracking search is implemented as option-returning generators. They may potentially return `None` if they fail to find a program. These can be a bit tricky to program. For example, when

$$\frac{\Delta;\Gamma \vdash e_1 : \tau_1 \,\&\, \varphi \qquad \Delta;\Gamma,(\mathtt{x} : \tau_1) \vdash e_2 : \tau_2 \,\&\, \varphi}{\Delta;\Gamma \vdash \mathtt{let}\ \mathtt{x} = e_1\ \mathtt{in}\ e_2 : \tau_2 \,\&\, \varphi}\ (\textsc{ELet})$$

$$\frac{\Delta;\Gamma \vdash e_0 : \mathtt{bool} \,\&\, \varphi \qquad \Delta;\Gamma \vdash e_1 : \tau \,\&\, \varphi \qquad \Delta;\Gamma \vdash e_2 : \tau \,\&\, \varphi}{\Delta;\Gamma \vdash \mathtt{if}\ e_0\ \mathtt{then}\ e_1\ \mathtt{else}\ e_2 : \tau \,\&\, \varphi}\ (\textsc{EIf})$$

Fig. 7. Type and effect rules for `let` and `if`

we introduced conditionals our first implementation attempt would simply perform three recursive generator calls, one for each of the branches and one for the test. As a consequence generation starting taking substantially more time. Only later did we regain some performance when we made the second and third recursive generator calls conditional on the success of the earlier ones.

6.1 Environment Representation

Each application of (ELam) (and (ELet)) introduces a new, locally bound variable. Hence we need to maintain an environment in order to determine the variables in scope along with their type. On the other hand, when we consider the possible applications of (EIndir) we seek variables based on their return types. Finally, after having chosen to generate an instance of (EIndir), we want to quickly locate a variable with the type associated with that instance.

In order to satisfy these demands we maintain three mappings: `env` from variables to types (implementing Γ), `revEnv` from types to (sets of) variables, and `returnEnv` from return types to (sets of) variables. Note that a function such as `f` : `unit` $\rightarrow$ `int` may occur several times in `returnEnv`: once with the key `int` and once with the key `unit` $\rightarrow$ `int`. In the presence of effects and sub-effecting a variable lookup in `revEnv` or `returnEnv` based on a type may not match exactly. We therefore normalize effects away in type signatures before inserting and looking up entries in `revEnv` and `returnEnv`, and only subsequently filter out variables with incompatible effect signatures (obtained from `env`). We collectively refer to this 3-way representation as the environment.

Our initial environment contains a number of bindings from OCaml's standard library. Whenever we apply (ELam) or (ELet) we need to extend the environment with the newly introduced variable $\mathtt{x} : \tau$ before passing it to a recursive generator call. We do so in three steps:

(1) First we update `env`, potentially shadowing an earlier binding of `x` as desired.
(2) Next we update `revEnv`. If another variable $\mathtt{x} : \tau'$ already exists in the environment we need to remove it from the set of variables associated to τ'. We subsequently add the new binding. If no other variables are associated with type τ we add a binding to a singleton variable set $\{\mathtt{x}\}$.
(3) Finally we update `returnEnv`. We do so by collecting the return types of τ and adding all of them following the above procedure.

6.2 Distribution

Rather than attempt to generate programs uniformly we skew the distribution in an attempt to generate programs that will stress the compiler backends. As traditional we do so by assigning integer weights to the generator's choice of rules. In our current implementation rule (EConst) is chosen with weight 6, instances of (EVar) are chosen with weight 1, rule (ELam) is chosen with weight 8, rule (EApp) is chosen with collective weight 8 (with weight 4 we pass the goal effect to the operator and with weight 4 we pass the goal effect to the operand), instances of (EIndir) are

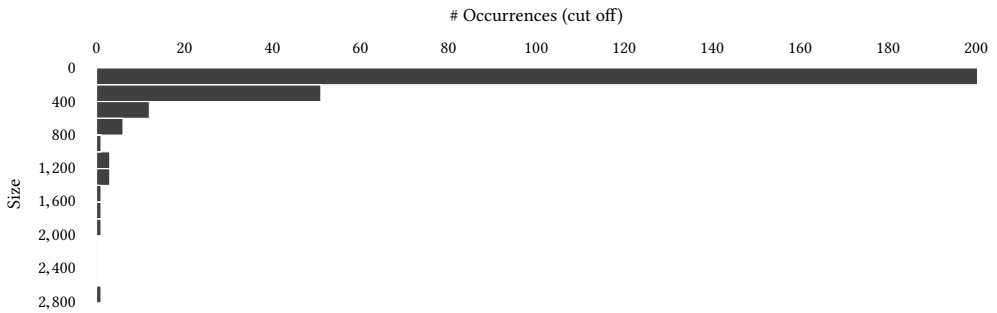
chosen with weight 4, rule (ELet) is chosen with weight 6, and rule (EIf) is chosen with weight 3. These frequencies do not represent the chance that a particular rule is chosen as a whole: for (EVar) the weight represents the chance of choosing a particular variable (with compatible type and effect). For rule (EIndir) the weight represent the chance that an instance is chosen, with one instance for each compatible type-and-effect signature in the environment.

This design of (EIndir) can be seen as an optimization of our backtracking generator in the style of Pałka et al. [2011]: If the generation of arguments fails for a function such as `(/) : int` $\xrightarrow{\texttt{ff/ff}}$ `int` $\xrightarrow{\texttt{tt/ff}}$ `int` for curried integer division it is likely that generation of arguments also fails for, e.g., function `(mod)` with the same type and effect signature. Therefore after collecting all variables with a compatible type and effect signature we group variables with identical signatures and add (EIndir) to the list of applicable rules once for each of these.

Our frequencies started out according to Pałka et al. [2011] and were later adjusted based on experiments. They will most likely be tweaked further in the future and clearly do not result in a uniform distribution. To observe the actual, resulting distribution arising from the above weights we defined a straight-forward measure of program size:

$$|\texttt{x}| = 1 \qquad |\texttt{let x} = e_1 \texttt{ in } e_2| = 1 + |e_0| + |e_1|$$
$$|\texttt{fun x -> } e| = 1 + |e| \qquad |\texttt{if } e_0 \texttt{ then } e_1 \texttt{ else } e_2| = 1 + |e_0| + |e_1| + |e_2|$$
$$|e_0\ e_1| = 1 + |e_0| + |e_1|$$

where in addition the size of literals $|c|$ is defined to be 1 for literals of base type and list length for list literals. Measured over a sample of 1000 arbitrary terms this yields an average program size of 64.4 with a standard deviation of 181.0, a median of 12, a minimum of 1, and a maximum of 2672. These statistics characterize a distribution centered around smaller programs (small average and median) albeit with occasional large ones (large maximum and large deviation). This characterization is further confirmed by visualizing the sample distribution, where we have cut off the histogram's x-axis and the top bar at 200 (there were 920 programs of size 1–199 in the sample):



6.3 Quickchecking the Testing Code

When testing requires additional programming there is a natural risk that the testing code itself contains bugs. Currently our testing module takes roughly 1000 lines of code. To reduce the risk of bugs in this code we have (naturally) tested it with QuickCheck. Separately from the "main test" of equivalence we therefore test:

- the underlying unification algorithm
- that our custom generator produces programs typeable by OCaml's type checker
- that our custom generator produces programs that are type-and-effect correct according to a type checker implementation of the type and effect system.

Thanks to these checks we have found and fixed a number of programming errors in our generator. In addition to the above, by classifying the output of a top-level call to our backtracking generator we have ensured that it succeeds in finding an evaluation-order independent program of type `int` in 1000 out of 1000 cases.

6.4 Limitations

Our approach comes with a number of limitations. Some of these are easier to address than others. In the easier end it would be natural to extend our approach with additional types (option types, chars, pairs, exceptions, user-defined datatypes, ...) and additional forms of expressions (pattern-matching, exception handlers, ...). Adding support for mutation based on reference cells comes with its own set of challenges, e.g., how many/deep combinations of the `ref` and `list` type constructors should one allow in a type generator? Presently we are also limited to programs that do not expect any input. In principle one could also generate a stream of input and supply this input to the generated program, e.g., by piping it on *standard in*. Finally we limit the current implementation to generating terminating programs: We do not attempt to generate recursive functions. We may however invoke recursive functions, e.g., over lists. Nothing fundamental about the approach prevents us from generating recursive functions by naming functions and recursively generating bodies with the function's own name and type-and-effect signature in scope. Since such programs are not guaranteed to terminate in general, one will instead have to run them with time outs akin to CSmith [Yang et al. 2011]. Compared to how compiler backends are traditionally tested (ensure that the backend produces correct code for an increasing collection of terminating, unit test programs) we do not see this last limitation as severe.

6.5 Shrinking Strategies

The generated counterexamples are often very large making it impossible to pinpoint what causes the observable output to differ. Therefore, we utilize shrinking in order to generate more comprehensible counterexamples. In QCheck, shrinking is implemented using iterators, that will lazily produce a sequence of possibly smaller counterexamples based on an initial counterexample. When a counterexample is found, QCheck traverses the sequence of candidates, and tests the property on every alternative. It stops at the first alternative that still proves to be a counterexample and then it starts the process over.

We formulate a range of possible shrinking strategies. Common to all of them is that they are type preserving. They are also effect-preserving in the sense of our preservation theorem: the explicitly type-and-effect annotated term resulting from each simplification step is validly effect annotated, however it may be overly conservative. In the end we converged on the following strategies, suitably phrased as a recursive traversal of the abstract syntax tree. We attempt to

(1) shrink literals with the builtin shrinkers for integers, strings, and list literals,
(2) replace any subterm (but literals) by a literal,
(3) replace unary and binary applications by their argument(s) (provided that the result type and the argument type agree),
(4) replace an immediate application by either the body of the lambda (provided the lambda-bound variable is free) or rewrite the application to a let-expression,
(5) pull let-binding in the operator position of an application out to surround the entire call,
(6) rewrite a let-binding to its body (provided that the let-bound variable is free),
(7) rewrite a let-bound let-binding to drop its outer binding (again provided that the let-bound variable is free),
(8) rewrite a conditional into either of its branches,

(9) wrap a conditional with a let-binding that serializes the potential effect of the condition, and
(10) always shrink subterms recursively

These strategies have developed from practical experiments and by observing opportunities for further cutting down a produced counterexample. The *constant replacement* strategy (2), half of (4) (a *β-reduction* strategy), and the *sub-term replacement* strategy incorporated in (3),(4), and (6) have been inherited from Pałka et al. [2011]. Rewriting an immediate application to a let-binding in (4) is specifically included to preserve a potential side effect, a situation which differs from the pure Haskell setting of Pałka et al. [2011]. All of the strategies may not produce a strictly smaller term, e.g., replacing a variable by a literal (2), the immediate application to let-binding in (4), or the pulling out of a let-binding in (5). They have nevertheless turned out to be effective in cooperation, e.g., the let-binding resulting from (4) may be located inside an application and subsequently be pulled out by (5), perhaps allowing a surrounding let-binding to be dropped by (7), if all uses of the variable has been replaced by literals (2).

During the development we have found that putting the most aggressive shrinking strategies early produces the most effective overall shrinker. This behaviour has a natural explanation, since attempts to, e.g., cut off entire sub-trees lets the resulting shrinker converge faster towards a locally minimal counterexample than if one insists that the shrinker attempts a number of smaller local rewrites first. Overall this experience confirms what seems to be established QuickCheck folklore.

7 EXPERIMENTS

We have set up a QuickCheck test with

- our type-and-effect-guided generator to generate programs of type and effect `int & tt/ff` and
- the property 'print the program to a file (wrapped in `let i = ... in print_int i` to ensure some output), compile the file with both backends, run both executables and compare output'

With this setup we have tested agreement between the two backends of OCaml version 4.02.3 and 4.04.0. By default our setup generates 500 arbitrary programs and tests them for agreement. Out of a sample of 20 subsequent runs testing OCaml version 4.04.0 we locate disagreements in 18 of them. The two completed runs take 2 minutes and 35 seconds and 3 minutes and 56 seconds on a normally loaded 2.8 GHz MacBook Pro laptop with 8GB of RAM. This makes for ∼0.39 seconds on average for generating an arbitrary program, writing it to file, compiling it with both backends, running both output programs, and *diff*'ing the outputs. The 18 runs that locate disagreements complete on average in 3 minutes and 55 seconds with a minimum of 22 seconds and a maximum of 6 minutes and 44 seconds. They invoke the generator between 18 and 490 times to do so. Note that the timings for these 18 runs include both the time for generating and testing 18–490 programs as well as the time spent shrinking the found counterexample.

We have found four new bugs and recreated two known ones using this approach. In all six cases, the error occurs in the optimizing native-code backend. In the first case from the introduction the native-code backend erroneously delays a side-effect in the operator of a partially applied function:

```
let k =
  (let i = print_newline ()
   in fun q -> fun i -> "") ()
in 0
```

This happens when the native-code backend constructs a closure representing the partially applied function. The backend does so according to the recipe described in the following comment (`asmcomp/closure.ml`, line 827):

```
We convert [f a] to [let a' = a in fun b c -> f a' b c]
                                          when fun_arity > nargs
```

This strategy carefully accounts for side effects in the evaluation of arguments such as `a`. The problem arises when the evaluation of `f` may have a side effect, in which case it will be delayed until the remaining arguments are supplied (if ever). In this case the correct strategy should be:

```
We convert [f a] to [let f' = f in
                     let a' = a in fun b c -> f' a' b c]
                                          when fun_arity > nargs
```

We have reported the bug and a patch has been approved for OCaml version 4.05.0.[2]

The next four bugs come in pairs of two for the division-related operators `/` and `mod` and are all connected to the native-code backend's handling of division by zero. In the first pair of bugs the native-code backend's optimizer erroneously removes a division by zero exception. This happens in connection with integer division when the divisor is sufficiently complex: [3]

```
(/) 0 (let e = not in pred 1)
```

In the second case this happens in connection with the integer modulo operator `mod`:

```
(mod) 0 (compare () ())
```

Both of these were fixed in connection with the report for the first (linked to in the above footnote).

The second pair of bugs is also related to the `/` and `mod` operators. In these two the native-code backend's optimizer erroneously removes the effect of the dividend preceding a division by zero exception. This issue may happen in connection with the division operator:

```
(/) (int_of_string "") (let e = let w = false in () in 0)
```

In this case the back end produces code that evaluates the divisor, tests the result for zero, and throws the exception, thereby removing the dividend's effect (a `Failure` exception). Similarly this may happen with the modulo operator:

```
(mod) (int_of_string "") (let m = print_int in 0)
```

We have also reported this pair of bugs and a patch is scheduled for OCaml version 4.05.0.[4]

In addition to these errors we have also found an already known bug similarly related to arithmetic optimization:

```
int_of_string (string_of_int (( * ) (int_of_string "") 0))
```

In this case the side-effect of the inner-most `int_of_string` call is erroneously optimized away. This counterexample also illustrates a limitation of our shrinker: it currently cannot shrink the above to the nested sub-expression `(( * ) (int_of_string "") 0)`. One way to do so would be to extend our shrinker with a general *sub-term replacement* strategy: attempt to replace an expression with one of its arbitrarily nested sub-expressions of compatible type if the free variables of the sub-expression are in scope in the outermost position. Some of our strategies in Sec. 6 already do so albeit only for immediate sub-expressions.

In order to better observe our compiler tester we instrumented it to print a period (`.`) whenever it tests the backends for agreement succesfully and print an `x` whenever they disagree. For runs that do not locate any disagreement this provides a low-level progress bar and for runs that do locate a disagreement this makes for a fascinating trail of shrinker attempts. Consider the following output revealing the above division issue:

[2] https://caml.inria.fr/mantis/view.php?id=7531

[3] The first of these two was initially shared with Jean-Christophe Filliâtre who then identified the root cause and reported the bug here: https://caml.inria.fr/mantis/view.php?id=7201. It was later recalled here: https://github.com/ocaml/ocaml/pull/954

[4] https://caml.inria.fr/mantis/view.php?id=7533

```
$ ./effmain.native -v
random seed: 379229741

.....................................................................
.................x.....x.....x.....x.....x......x......x.......x......
.x.x......x......x.......x......
law bytecode/native backends agree: 90 relevant cases (90 total)
  test `bytecode/native backends agree`
  failed on ≥ 1 cases:
  Some ((mod) (int_of_string "") (let m = print_int in 0))
                                                  (after 12 shrink steps)

failure (1 tests failed, ran 1 tests)
```

The first `x` above marks the point where QCheck first identifies a counterexample. The subsequent mixed trail of periods and `x`s illustrates how shrinking systematically attempts to cut down the counterexample to something that remains a counterexample.

8 RELATED WORK

We separate the discussion of related work in two: one part concerns related work on type and effect systems and one part concerns related work on randomized testing.

8.1 Effect Analyses

One particular kind of effect is assignment. Classical *def-use* data analyses [Banning 1979; Nielson et al. 1999] can answer whether a piece of code affects mutable variables with assignment. In comparison our type and effect system can express assignments as effects and thereby answer evaluation-order dependence involving references and assignment. In doing so we cannot express assignment as precisely: once we inject an assignment as an effect we lose track of the particular variable being assigned. Type and effect systems as an independent discipline started with the seminal works of Gifford and Lucassen [1986; 1988]. These were concerned with determining the effects (assignment, allocation, ...) of sub-expressions with the goal of driving a parallelising compiler.

The Java programming language [Gosling et al. 2000] comes with a built-in effect analysis for exceptions: a Java method that may throw a *checked exception* has to be annotated to do so, otherwise it will be rejected by the compiler. Java has no such requirement for *unchecked exceptions* such as `NullPointerException`. Leroy and Pessaux [2000] develop a type-based analysis of uncaught exceptions for OCaml with obvious parallels to our type and effect system: since exceptions are an effect they are also expressible in our framework, albeit again not as precisely: once we inject an exception as an effect we lose track of the particular exception that may be thrown. On the other hand our type and effect system can capture other kinds of effects, e.g., output. For a brief survey of type and effect systems we refer to Nielson and Nielson [1999] and for a more in depth introduction to the area we refer to the textbook of Amtoft et al. [1999].

Astreé [Blanchet et al. 2003] is a general static analysis for a large subset of C that can detect a number of cases leading to C's notorious *undefined behaviours*. Astreé is well known for its precision: in some cases it has no false alarms, yet it computes a conservative over-approximation of the input program's behaviour. Again our aim with our type and effect analysis is different: we seek to develop a simple approach that lets us generate programs free of evaluation-order dependence, not develop a precise analysis for detecting *undefined behaviour*.

8.2 Randomized Testing

CSmith [Yang et al. 2011] is a generator of programs free of C's notorious undefined behaviour. By doing so it can be used to test C compilers against each other (or different optimization levels of same C compiler against each other), and it has been used to find hundreds of bugs in GCC, LLVM, etc. CSmith supports an impressive range of language features compared to our implementation. In our approach we have taken a different route to developing our generator than the more ad hoc approach of Yang et al. [2011], by first formalizing and proving sound an effect system to drive our generator. Admittedly this formal route may be easier for a language with roots in typed lambda calculus such as OCaml, than for a language such as C with a complex textual specification containing many tricky cases of undefined behaviour.

Le et al. [2014] develop an *equivalence modulo inputs* (EMI) methodology as a form of *differential testing* for testing C compilers. Rather than attempt to generate arbitrary programs from scratch Le et al. [2014] instead take an existing collection of programs, pick one and manipulate it in a manner that is meaning preserving for a particular input, e.g., permuting code in a branch which is not executed for the particular input. If none of the input programs contain *undefined behaviour* their manipulations will not give rise to undefined behaviour for a run with the particular input. The Orion tool realizing the approach has successfully found an impressive range of bugs in GCC and LLVM. The EMI approach allows one to test equivalence of two related programs compiled with *the same compiler*, which is fundamentally different than our comparison of *two different compiler backends* (with one acting as oracle for the other).

Pałka et al. [2011] originally developed the goal-directed program generation approach to test the Haskell compiler's strictness analyzer [Pałka et al. 2011]. Since side-effects in Haskell are captured in the type system, e.g., by monads, they did not have the issue of impure programs with evaluation-order dependence: in Haskell the order of side effects would be dictated, e.g., by the monadic operations.

Claessen et al. [2015] have developed an approach to automatically construct generators for constrained random data types that have a provably uniform distribution. The focus of the present work is different in that we have developed formal constraints (in the form of a type and effect system) for testing language implementations for agreement up to evaluation order. We make no formal claims of uniformity but in future work we would like to investigate and integrate the approaches of Claessen et al. [2015] in order to do so.

The goal-directed generation approach from inference rules has since been applied in the context of PLT Redex to generically test type systems defined by such rules [Fetscher et al. 2015]. Again the focus of this work is different: Fetscher et al. [2015] seek to develop a general approach for type systems formalized in PLT Redex whereas we formalize and develop a generator with a specific purpose. It would be interesting to investigate whether the PLT Redex inference-driven generator can handle our type and effect system.

St-Amour and Toronto [2013] use property-based testing to check the soundness of Typed Schemes initial typing environment. In an advanced type system such as Typed Scheme it is a laborious and error-prone task to provide sound type signatures for all the builtin primitives in the initial environment. St-Amour and Toronto [2013] show how QuickCheck techniques can provide a fruitful quality control for this task. We could use the techniques of St-Amour and Toronto [2013] to ensure the soundness of the initial type (and effect) environment that drives our generator.

SmartCheck [Pike 2014] is an extension of QuickCheck that tries to generalize failed counterexamples based on their shape in order to narrow the problem further down to the developer. Over the course of testing the OCaml backends we have repeatedly found variations of the same bug, e.g., compare the following two shrunk counterexamples produced by our testing tool:

```
let k =
  (let i = print_newline ()
   in fun q -> fun i -> "") ()
in 0
```

```
let s =
  (let c = string_of_int (int_of_string "")
   in fun d -> fun l -> fun f -> 0) false
in 0
```

A SmartCheck implementation for OCaml would be helpful in identifying the common structure of these two counterexamples: an operator with a side-effect that is partially applied, erroneously has its side-effect delayed indefinitely by the native code backend. Doing so could involve generalizing not only syntax, but also types and effects. Alternatively we could apply the approach of Hughes [2016] to specify or document the buggy behaviour in the QuickCheck tests, and thereby let the bug search continue unaffected. Hughes [2016] reports about how doing so became a necessity as part of testing the AUTOSAR components underlying Volvo cars, as the QuickCheck testers would otherwise have to wait for developers (from one of Volvo's suppliers) to fix the identified errors. This experience most recently motivated the development of MoreBugs [Hughes et al. 2016]: a QuickCheck framework engineered to avoid the repeated rediscovery of the same bugs. The idea of MoreBugs is the same as SmartCheck: it attempts to generalize bug patterns. However the goal is different, namely to use the bug patterns as a feedback mechanism to adjust the generators so as to spend the generation time wisely on finding new, undiscovered bugs. Again our approach could benefit from a MoreBugs implementation and again this could involve generalizing not only syntax, but also types and effects.

9 CONCLUSION AND FUTURE WORK

We have presented a novel type and effect system for driving a goal-directed program generator. We have implemented the approach and applied it to test the two official OCaml compiler backends for agreement. In doing so we have found a number of disagreements on subtle cornercase of a core language. Overall we believe the approach illustrates the strength of QuickCheck and is promising for quality assurance of functional language processors.

We envision a range of directions for future work. One direction would be to generalize the type and effect system (a) to a polymorphic system, (b) to include effect variables to express relations between side-effects in signatures, or (c) to refine it to more precisely express the kind of run-time effect a sub-expression may have. A second direction would be to extend the supported language with additional expression constructs and types. A third direction would be to utilize the approach to test additional language implementations. For OCaml alone, several such implementations exist (js_of_ocaml, Bloomberg's BuckleScript OCaml-to-JavaScript compiler, OCaml-Java, ocamlcc, ...). By retargeting the approach to, e.g., Standard ML, it could be used to test some of its implementations (SML/NJ, MLton, Moscow ML, PolyML, MLkit, SML#, ...) against each other. In particular, Moscow ML is based on OCaml's right-to-left evaluating bytecode interpreter and would therefore deviate from the formally defined left-to-right evaluation order of Standard ML. One could in principle also use the approach to test implementations of a dynamically typed functional language such as Scheme. In Scheme the evaluation order is also undefined and several implementations exist (Bigloo, Chez, Chicken, Gambit, Racket/MzScheme, ...). Although driven by types and effects, after a type-and-effect erasure (and modulo an initial environment) all generated programs should however be valid. Such a setup would strictly speaking not target all possible Scheme programs, but only the subset of programs that type (and effect) check statically. Nevertheless it would be an easy way to cheaply test agreement of a range of implementations on an unbounded number of programs.

A APPENDIX

The source code of the prototype tester is available for download: https://github.com/jmid/efftester/.

A precompiled prototype artifact is also available as a Docker image:

https://hub.docker.com/r/jmid/ocaml-efftester/

The full proofs are available from the authors' full version:

http://janmidtgaard.dk/papers/Midtgaard-al:ICFP17-full.pdf

ACKNOWLEDGMENTS

We thank the anonymous ICFP reviewers for their constructive feedback. This work was supported by IDEA4CPS, granted by the Danish National Research Foundation (DNRF86-10).

REFERENCES

Torben Amtoft, Hanne Riis Nielson, and Flemming Nielson. 1999. *Type and effect systems - behaviours for concurrency.* Imperial College Press.

John Banning. 1979. An Efficient Way to Find Side Effects of Procedure Calls and Aliases of Variables. In *Proceedings of the Sixth Annual ACM Symposium on Principles of Programming Languages*, Barry K. Rosen (Ed.). San Antonio, Texas, 29–41.

Bruno Blanchet, Patrick Cousot, Radhia Cousot, Jérôme Feret, Laurent Mauborgne, Antoine Miné, David Monniaux, and Xavier Rival. 2003. A static analyzer for large safety-critical software. In *Proceedings of the ACM SIGPLAN 2003 Conference on Programming Languages Design and Implementation*, Ron Cytron and Rajiv Gupta (Eds.). San Diego, California, 196–207.

Koen Claessen, Jonas Duregård, and Michal H. Pałka. 2015. Generating constrained random data with uniform distribution. *Journal of Functional Programming* 25 (2015).

Koen Claessen and John Hughes. 2000. QuickCheck: A Lightweight Tool for Random Testing of Haskell Programs. In *Proceedings of the Fifth ACM SIGPLAN International Conference on Functional Programming (ICFP'00)*, Philip Wadler (Ed.). Montréal, Canada, 53–64.

Patrick Cousot and Radhia Cousot. 1977. Abstract Interpretation: a unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *Proceedings of the Fourth Annual ACM Symposium on Principles of Programming Languages*, Ravi Sethi (Ed.). Los Angeles, California, 238–252.

Burke Fetscher, Koen Claessen, Michal H. Pałka, John Hughes, and Robert Bruce Findler. 2015. Making Random Judgments: Automatically Generating Well-Typed Terms from the Definition of a Type-System. In *Programming Languages and Systems, 24th European Symposium on Programming, ESOP 2015 (Lecture Notes in Computer Science)*, Jan Vitek (Ed.), Vol. 9032. Springer-Verlag, 383–405.

David K. Gifford and John M. Lucassen. 1986. Integrating Functional and Imperative Programming. In *Proceedings of the 1986 ACM Conference on Lisp and Functional Programming*, William L. Scherlis and John H. Williams (Eds.). Cambridge, Massachusetts, 28–38.

James Gosling, Bill Joy, Guy Steele, and Gilad Bracha. 2000. *Java Language Specification, Second Edition: The Java Series* (2nd ed.). Addison-Wesley, Boston, MA, USA.

John Hughes. 2016. Experiences with QuickCheck: Testing the Hard Stuff and Staying Sane. In *A List of Successes That Can Change the World - Essays Dedicated to Philip Wadler on the Occasion of His 60th Birthday (Lecture Notes in Computer Science)*, Sam Lindley, Conor McBride, Philip W. Trinder, and Donald Sannella (Eds.), Vol. 9600. Springer-Verlag, 169–186.

John Hughes, Ulf Norell, Nicholas Smallbone, and Thomas Arts. 2016. Find more bugs with QuickCheck!. In *Proceedings of the 11th International Workshop on Automation of Software Test, AST@ICSE 2016, Austin, Texas, USA, May 14-15, 2016*, Christof J. Budnik, Gordon Fraser, and Francesca Lonetti (Eds.). ACM, 71–77.

Patrick Kasting and Mathias Nygaard Justesen. 2016. *Quickchecking OCaml compilers by generating lambda terms.* Unpublished course project report. Technical University of Denmark, Lyngby, Denmark.

Vu Le, Mehrdad Afshari, and Zhendong Su. 2014. Compiler validation via equivalence modulo inputs. In *Proceedings of the ACM SIGPLAN 2014 Conference on Programming Languages Design and Implementation, PLDI'14*, Michael F. P. O'Boyle and Keshav Pingali (Eds.). 216–226.

Xavier Leroy. 1990. *The Zinc experiment: an economical implementation of the ML language.* Rapport Technique 117. INRIA Rocquencourt, Le Chesnay, France.

Xavier Leroy and François Pessaux. 2000. Type-based analysis of uncaught exceptions. *ACM Transactions on Programming Languages and Systems* 22, 2 (2000), 340–377.

John M. Lucassen and David K. Gifford. 1988. Polymorphic effect systems. In *Proceedings of the Fifteenth Annual ACM Symposium on Principles of Programming Languages*, Jeanne Ferrante and Peter Mager (Eds.). ACM Press, San Diego, California, 47–57.

John C. Martin. 1997. *Introduction to Languages and the Theory of Computation.* McGraw-Hill.

Flemming Nielson and Hanne Riis Nielson. 1999. Type and Effect Systems. In *Correct System Design, Recent Insight and Advances, (to Hans Langmaack on the occasion of his retirement from his professorship at the University of Kiel) (Lecture Notes in Computer Science)*, Ernst-Rüdiger Olderog and Bernhard Steffen (Eds.), Vol. 1710. Springer-Verlag, 114–136.

Flemming Nielson, Hanne Riis Nielson, and Chris Hankin. 1999. *Principles of Program Analysis.* Springer.

Michal H. Pałka, Koen Claessen, Alejandro Russo, and John Hughes. 2011. Testing an optimising compiler by generating random lambda terms. In *Proceedings of the 6th International Workshop on Automation of Software Test, AST 2011*. 91–97.

Benjamin C. Pierce. 2002. *Types and Programming Languages.* The MIT Press.

Lee Pike. 2014. SmartCheck: automatic and efficient counterexample reduction and generalization. In *Proceedings of the 2014 ACM SIGPLAN symposium on Haskell, Gothenburg, Sweden, September 4-5, 2014*, Wouter Swierstra (Ed.). 53–64.

Vincent St-Amour and Neil Toronto. 2013. Experience Report: Applying Random Testing to a Base Type Environment. In *Proceedings of the 18th ACM SIGPLAN International Conference on Functional Programming (ICFP'13)*, Greg Morrisett and Tarmo Uustalu (Eds.). Boston, MA, 351–356.

Andrew K. Wright and Matthias Felleisen. 1994. A syntactic approach to type soundness. *Information and Computation* 115 (1994), 38–94.

Xuejun Yang, Yang Chen, Eric Eide, and John Regehr. 2011. Finding and Understanding Bugs in C Compilers. In *Proceedings of the ACM SIGPLAN 2011 Conference on Programming Languages Design and Implementation, PLDI'11*, David Padua (Ed.). San Jose, California, 283–294.